Secure Computation of Hidden Markov Models

Mehrdad Aliasgari and Marina Blanton
Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA
maliasga@nd.edu, mblanton@nd.edu

Keywords: Secure Computation, Floating Point, Hidden Markov Models, Gaussian Mixture Models

Abstract: Hidden Markov Model (HMM) is a popular statistical tool with a large number of applications in pattern recognition. In some of such applications, including speaker recognition in particular, the computation involves personal data that can identify individuals and must be protected. For that reason, we develop privacy-preserving techniques for HMM and Gaussian mixture model (GMM) computation suitable for use in speaker recognition and other applications. Unlike prior work, our solution uses floating point arithmetic, which allows us to simultaneously achieve high accuracy, provable security guarantees, and reasonable performance. We develop techniques for both two-party HMM and GMM computation based on threshold homomorphic encryption and multi-party computation based on threshold linear secret sharing, which are suitable for secure collaborative computation as well as secure outsourcing.

1 INTRODUCTION

Hidden Markov Models (HMMs) have been an invaluable and widely used tool in the area of pattern recognition. They have applications in bioinformatics, credit card fraud detection, intrusion detection, communication networks, machine translation, cryptanalysis, robotics, and many other areas. An HMM is a powerful statistical tool for modeling sequences that can be characterized by an underlying Markov process with unobserved (or hidden) states, but visible outcomes. One important application of HMMs is voice recognition, which includes both speech and speaker recognition. For both, HMMs are the most common and accurate approach, and we use this application as a running example that guides the computation and security model for this work.

When an HMM is used for the purpose of speaker recognition, usually one party supplies a voice sample and the other party holds a description of an HMM that represents how a particular individual speaks and processes the voice sample using its model and the corresponding HMM algorithms. Security issues arise in this context because one's voice sample and HMMs are valuable personal information that must be protected. In particular, a server that stores hidden Markov models for users is in possession of sensitive biometric data, which once leaked to insiders or outsiders can be used to impersonate the users. For that reason, it is desirable to minimize exposure of voice samples and HMMs corresponding to an individual when such data are being used for authentication or other purposes. To this end, in this work we design solutions for securely performing computation on HMMs in such a way that no information about private data is revealed as a result of execution other than the agreed upon output. This will immediately imply privacy-preserving techniques for speaker recognition as well as other applications of HMMs.

There are three different types of problems and corresponding algorithms for HMM computation: the Forward algorithm, the Viterbi algorithm, and the Expectation Maximization (EM) algorithm. Because the Viterbi algorithm is most commonly used in voice recognition, we develop a privacy-preserving solution for that algorithm, but our techniques can be used to securely execute other HMM algorithms as well. Furthermore, to ensure that Gaussian mixture models (GMMs), which are commonly used in HMM computation, can be part of secure computation as well, we integrate GMM computation in our privacy-preserving solution.

One significant difference between our and prior work is that, unlike other publications, we develop techniques for computation on floating point numbers which provide adequate precision and are most appropriate for HMM computation. We also do not compromise on security, and all of the techniques we develop are provably secure under standard and rigorous security models, while at the same time providing

reasonable performance.

To cover as wide of a range of application scenarios as possible, we consider multiple settings for HMM computation: (i) the two-party setting in which a client interacts with a server and (ii) the multi-party setting in which the HMM computation is carried out by $n > 2$ parties, which is suitable for joint computation by several participants as well as secure outsourcing of HMM computation to multiple servers by one or more computationally limited clients.

To summarize, our contributions consist of developing provably secure HMM and GMM computation techniques based on Viterbi algorithm using floating point arithmetic. Our techniques are suitable for two-party and multi-party computation in a variety of settings and are designed with their efficiency in mind.

2 RELATED WORK

To the best of our knowledge, privacy-preserving HMM computation was first considered in (Smaragdis and Shashanka, 2007) which provides a secure two-party solution for speech recognition using a homomorphic encryption scheme on integer domain. Unfortunately, integer representation is not sufficient for HMM computation, and this work failed to address non-integer values. In particular, HMM computation involves various operations on probability values which occupy a large range of real numbers and demand high precision.

Nevertheless, the techniques provided in (Smaragdis and Shashanka, 2007) were later used as-is in (Shashanka, 2010) for Gaussian mixture models. The same idea was used in (Pathak et al., 2011; Pathak et al., 2012) to develop privacy-preserving speaker verification for joint two-party computation, where the HMM parameters were stored in an encrypted domain. Similar to (Shashanka, 2010), the goal of (Pathak and Raj, 2011) was to provide secure two-party GMM computation using the same high-level idea but with implementation differences, yet their solution has security weaknesses. In particular, the proposed Logsum protocol reveals a non-trivial amount of information about the private inputs, which, in combination with other computation or outside knowledge, may allow for full recovery of the inputs. We provide more detail regarding this security weakness in Appendix 7. All the above work addresses integer-based solutions in the two-party setting. In addition, some of the above techniques were used in privacy-preserving network analysis and anomaly detection in a joint two-party (or multi-party) computation (Nguyen and Roughan, 2012b; Nguyen and Roughan, 2012a). Clearly, there is need to develop secure computation techniques for HMMs on standard real number representations with provable security. In particular, integer or fixed point representations have at least two disadvantages: They demand substantially larger bit length representation than could be used otherwise and the representation error can accumulate and introduce fatal inaccuracies. Subsequently, integer or fixed-point representations become impractical for applications that require high precisions such as HMM computations. To address this, Franz et al. (Franz et al., 2012) proposed solutions for secure HMM forward algorithm computation in the two-party setting using logarithmic representation of real numbers. The logarithmic representation, although better than fixed point or integer representation, still fails to cope with the vast range of real numbers used in HMMs. The look-up tables in (Franz et al., 2012) grow exponentially in the bitlength of the operands. In fact, in practice HMMs are run on floating point numbers to avoid the difficulties mentioned above. Therefore, in this work we propose the first provably secure floating point computation for HMM algorithms with reasonable performance.

3 HIDDEN MARKOV MODELS AND GAUSSIAN MIXTURE MODELS

A Hidden Markov Model (HMM) in a statistical model that follows the Markov property (where the transition at each step depends only on the previous transition) with hidden states, but visible outcomes. The inputs are a sequence of observations, and for each sequence of observations (or outcomes), the computation consists of determining a path of state transitions which is the likeliest among all paths that could produce the given observations. More formally, an HMM consists of:

- N states $S_1, \cdots, S_N$;
- M possible outcomes $m_1, \cdots, m_M$;
- a vector $\pi = \langle \pi_1, \ldots, \pi_N \rangle$ that contains the initial state probability distribution, i.e., $\pi_i = \Pr[q_1 = S_i]$, where q is a random variable over the set of states indexed by the transition number;
- a matrix A of size $N \times N$ that contains state transition probabilities, i.e., a cell a_{ij} of A at row i and column j contains the probability of the transition from state S_i to state S_j $a_{ij} = \Pr[q_{k+1} = S_j \mid q_k = S_i]$;

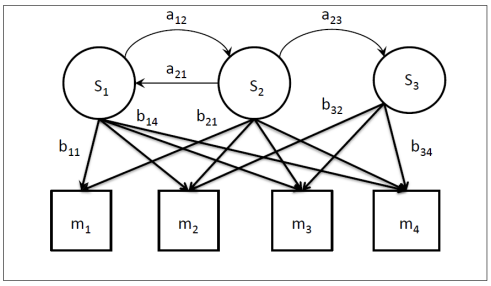

Figure 1: An example of a hidden Markov model with s_i's representing states and m_i's representing outcomes.

- a matrix B of size $N \times M$ that contains output probabilities, i.e., a cell b_{ij} of B at row i and column j contains the probability of state S_i outputting outcome m_j $b_{ij} = \Pr[q_k = S_i \mid X_k = m_j]$.

In the above, observations $X_1, \ldots, X_T$ form HMM's input, to which we collectively refer as X. The above parameters define an HMM. In our running application of speaker recognition, an HMM is a model that represents how a particular person speaks and an input corresponds to the captured voice sample of a client. Figure 1 shows an example of an HMM.

In most cases, matrix B should be computed based on observations. Usually this is done by evaluation of the observed value on probability distributions of states' outcomes. One very common distribution model is a Gaussian Mixture Model (GMM) which is used to compute the elements of matrix B. GMMs are mixtures of Gaussian distributions that represent the overall distribution of observations. Namely, an observation is evaluated on a number of Gaussian distributions with different parameters and the evaluations are combined together to produce the final probability of a random variable acquiring that particular observation. In the case of HMMs, we use a GMM to compute the output probability of state S_j producing an observation at time k as follows:

$$b_{jk} = \sum_{i=1}^{\alpha} w_i e^{-\frac{1}{2}(X_k-\mu_i)^T \Sigma_i^{-1}(X_k-\mu_i)} \qquad (1)$$

In the above, X_k is a vector of size f that represents the random variable corresponding to the observation at time k. In voice applications, X_k usually contains the Mel-frequency cepstral coefficients (MFCCs). The parameter α is the total number of mixture components (here, Gaussian distributions). The ith component has a mean vector μ_i of size f and a covariance matrix Σ_i of size $f \times f$. The components are added together, each weighted by a mixture weight w_i, to produce the probability distribution of state S_j when the observed random variable is X_k. We use notation μ, Σ, and w to refer to the sequence of μ_i, Σ_i, and w_i, respectively, for $i = 1, \ldots, \alpha$.

There are three different types of problems and respective dynamic-programming algorithms for HMM computation: the Forward Algorithm, the Viterbi Algorithm, and the EM (Expectation Maximization) Algorithm (Rabiner, 1989). In the Forward Algorithm, the goal is to compute the probability of each state for each transition given a particular sequence of observations. Namely, in this algorithm, we compute $\Pr[q_k = S_i \mid X_1 \cdots X_T]$. In the Viterbi algorithm, after observing the outcomes, the goal is to construct the path of states which is the most likely among all possible paths that can produce the observations, as well as the probability of the most likely path. In other words, for any given sequence of observations, each path has a certain probability of producing that sequence of observations. The output of Viterbi algorithm is the path with the highest probability and the value of the probability. The computation performed in Viterbi algorithm uses Forward algorithm. In the EM algorithm, the goal is to learn the HMM. Namely, given a sequence of observations, this algorithm computes the parameters of the most likely HMM that could have produced this sequence of observations. All of these three algorithms use dynamic programming techniques and have complexity of $O(TN^2)$.

Because in this work we use speaker recognition to demonstrate secure techniques for HMM computation, we focus on the Viterbi algorithm used in speaker recognition. The techniques developed in this work, however, can also be used to construct secure solutions for the other two algorithm. In what follows, we provide a brief description of the Viterbi algorithm and refer the reader to online materials for the Forward and EM algorithms. In the algorithm below, P^* is the probability of the most likely path for a given sequence of observations and $q^* = \langle q_1^*, \ldots, q_T^* \rangle$ denotes the most likely path itself. The computation uses dynamic programming to store intermediate probabilities in δ, after which the path of the maximum likelihood is computed and placed in q^*.

$\langle P^*, q^* \rangle \leftarrow \mathsf{Viterbi}(\lambda = \langle N, T, \pi, A, B \rangle)$

1. Initialization Step: for $i = 1$ to N do
   (a) $\delta_1(i) = \pi_i b_{i1}$
   (b) $\psi_1(i) = 0$
2. Recursion Step: for $k = 2$ to T and $j = 1$ to N do
   (a) $\delta_k(j) = \left(\max_{1 \le i \le N} [\delta_{k-1}(i) a_{ij}] \right) b_{jk}$
   (b) $\psi_k(i) = \arg\max_{1 \le i \le N} [\delta_{k-1}(i) a_{ij}]$
3. Termination Step:
   (a) $P^* = \max_{1 \le i \le N} [\delta_T(i)]$

(b) $q_T^* = \arg\max_{1 \le i \le N} \delta_T(i)$

(c) for $k = T-1$ to 1 do $q_k^* = \psi_{k+1}(q_{k+1}^*)$

4. Return $\langle P^*, q^* \rangle$

In speaker recognition, we apply the Viterbi algorithm to extracted voice features and an HMM that was created using a GMM and training voice features. The overall computation then consists of forming an HMM using GMM computation and executing the Viterbi algorithm, as given next.

$\langle P^*, q^* \rangle \leftarrow \mathsf{HMM}(N, T, \pi, A, \alpha, w, \mu, \Sigma, X)$

---

1. For $j = 1$ to N and $k = 1$ to T, compute b_{jk} as in equation 1 using α, w_i's, μ_i's, Σ_i's, and X_k.
2. Set $\lambda = \langle N, T, \pi, A, B \rangle$.
3. Execute $\langle P^*, q^* \rangle = \mathsf{Viterbi}(\lambda)$.
4. Return $\langle P^*, q^* \rangle$.

---

4 FRAMEWORK

In this section, we introduce two categories of secure computation that we consider in this work, precisely define the computation to be carried out, and the security model for each respective type of secure computation.

The first category of secure computation that we consider is secure *two-party* computation. Without loss of generality, we will refer to the participants as the client and the server. Using speaker recognition as the example application, the client possesses a voice sample, the server stores a model that represents how a registered user speaks, and user authentication is performed by conducting HMM computation on the client's and server's inputs. Therefore, for the purposes of this work, we assume that the client owns the observations to an HMM, i.e., $X_1, \ldots, X_T$, and the server holds the parameters of the HMM and GMM, i.e., N, vector π, matrix A, α, mixture weights w, vectors μ, and matrices Σ. Because even the parameters of HMM might reveal information about the possible input observations, to build a fully privacy-preserving solution in which the server does not learn information about user biometrics, the server should not have access to the HMM parameters in the clear. For that reason, we assume that the server holds the parameters π, A, B, w, μ, and Σ in an encrypted form. Then to permit the computation to take place on encrypted data, we resort to an encryption scheme with special properties, namely, semantically secure additively homomorphic public-key encryption scheme (defined below). Furthermore, to ensure that neither the server can decrypt the data it stores, nor the (untrusted) client can decrypt the data (or a function thereof) without the server's consent, we utilize a (2, 2)-threshold encryption scheme. Informally, it means that the decryption key is partitioned between the client and the server, and each decryption requires that both of them participate. This means that the client and the server can jointly carry out the HMM computation, and make the result available to either or both of them. For concreteness of our description, we will assume that the server learns the outcome.

A public-key encryption scheme is defined by three algorithms Gen, Enc, Dec, where Gen is a key generation algorithm that on input a security parameter κ produces a public-private key pair (pk, sk); Enc is an encryption algorithm that on input a public key pk and message m produces ciphertext c; and Dec is a decryption algorithm that on input private key sk and ciphertext c produces decrypted message m or special character $\perp$ that indicates failure. For conciseness, we use notation $\mathsf{Enc}_{pk}(m)$ and $\mathsf{Dec}_{sk}(c)$ in place of $\mathsf{Enc}(pk, m)$ and $\mathsf{Dec}(sk, c)$, respectively. An encryption scheme is said to be additively homomorphic if applying an operation to two ciphertexts results in the addition of the messages that they encrypt, i.e., $\mathsf{Enc}_{pk}(m_1) \cdot \mathsf{Enc}_{pk}(m_2) = \mathsf{Enc}(m_1 + m_2)$. This property also implies that $\mathsf{Enc}_{pk}(m)^k = \mathsf{Enc}_{pk}(k \cdot m)$ for a known k. In a public-key (n,t)-threshold encryption scheme, the decryption key sk is partitioned among n parties, and $t \le n$ of them are required to participate in order to decrypt a ciphertext. Lastly, a semantically secure encryption scheme guarantees that no information about the encrypted message can be learned from its ciphertext with more than a negligible (in κ) probability. Semantically secure additively homomorphic threshold public-key encryption schemes are known, one example of which is Pailler encryption (Paillier, 1999).

We obtain that in the two-party setting, the client and the server share the decryption key to a semantically secure additively homomorphic (2, 2)-threshold public-key encryption scheme. The client has private input $X_1, \ldots, X_T$ and its share of the decryption key sk; the server has input $\mathsf{Enc}_{pk}(\pi_i)$ for $i \in [1, N]$, $\mathsf{Enc}_{pk}(a_{ij})$ for $i \in [1, N]$ and $j \in [1, N]$, $\mathsf{Enc}_{pk}(w_i)$ for $i \in [1, \alpha]$, encryption of each element of μ_i and Σ_i for $i \in [1, \alpha]$, and its share of sk. The computation consists of executing the Viterbi algorithm on their inputs, at the end of which the server learns P^* and q_i^* for $i = 1, \ldots, T$. The size of the problem, i.e., parameters N, T, α, and f, are assumed to be known to both parties.

The second category of secure computation that

we consider is secure *multi-party* computation on HMMs. In this setting, either a number of parties hold inputs to a multi-observer HMM or one or more clients wish to outsource HMM computations to a collection of servers. More generally, we divide all participants into three groups: (i) the input parties who collectively possess the private inputs, (ii) the computational parties who carry out the computation, and (iii) the output parties who receive the result(s) of the computation. These groups can be arbitrarily overlapping, which gives great flexibility in the setup and covers all possible cases of joint multi-party computation (where the input owners carry out the computation themselves, select a subset of them, or seek help of external computational parties) and outsourcing scenarios (by either a single party or multiple input owners).

To conduct computation on protected values in this setting, we utilize an information-theoretically secure threshold linear secret sharing scheme (such as Shamir secret sharing scheme (Shamir, 1979)). In a (n,t)-threshold secret sharing scheme, a secret value s is partitioned among n participants in such a way that the knowledge of t or fewer shares information-theoretically reveals no information about s, while $t+1$ or more shares allow for efficient reconstruction of s. Such schemes avoid the use of computationally expensive public-key encryption techniques and instead operate on small integers (in a field $\mathbb{F}_p$, normally with prime p) of sufficient size to represent all values. In a linear secret sharing scheme, any linear combination of secret sharing values (which in particular includes addition and multiplication by a known constant) is performed by each participant locally, while multiplication requires interaction of all parties. It is usually required that $t < n/2$ and therefore the number of computational parties $n > 2$.

We then obtain that in this setting the input parties share their private inputs among $n > 2$ computational parties, the computational parties execute the Viterbi algorithm on secret-shared values, and communicate shares of the result to the output parties, who reconstruct the result from their shares. As before, the size of the problem, namely, the parameters N, T, α, and f, is known to all parties.

We next formally define security using the standard definition in secure multi-party computation for semi-honest (also known as honest-but-curious or passive) adversaries, i.e., those that follow the computation as prescribed, but might attempt to learn additional information about the data from the intermediate results. We show our techniques secure in the semi-honest model. Standard techniques for making the computation robust to malicious behavior, where the participants can arbitrarily deviate from the prescribed computation, apply to our protocols as well.

Definition 1. *Let parties $P_1,\ldots,P_n$ engage in a protocol Π that computes function $f(\mathsf{in}_1,\ldots,\mathsf{in}_n) = (\mathsf{out}_1,\ldots,\mathsf{out}_n)$, where in_i and out_i denote the input and output of party P_i, respectively. Let $\mathrm{VIEW}_\Pi(P_i)$ denote the view of participant P_i during the execution of protocol Π. More precisely, P_i's view is formed by its input and internal random coin tosses r_i, as well as messages $m_1,\ldots,m_k$ passed between the parties during protocol execution*

$$\mathrm{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k).$$

Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ denote a subset of the participants for $t < n$ and $\mathrm{VIEW}_\Pi(I)$ denote the combined view of participants in I during the execution of protocol Π (i.e., the union of the views of the participants in I). We say that protocol Π is t-private in presence of semi-honest adversaries if for each coalition of size at most t there exists a probabilistic polynomial time simulator S_I such that

$$\{S_I(\mathsf{in}_I, f(\mathsf{in}_1,\ldots,\mathsf{in}_n)\} \equiv \{\mathrm{VIEW}_\Pi(I), \mathsf{out}_I\},$$

where $\mathsf{in}_I = \bigcup_{P_i \in I}\{\mathsf{in}_i\}$, $\mathsf{out}_I = \bigcup_{P_i \in I}\{\mathsf{out}_i\}$, and $\equiv$ denotes computational or statistical indistinguishability.

In the two-party setting, we have that $n = 2$, $t = 1$, and the participants' inputs in_1, in_2 and outputs out_1, out_2 are set as described above. In the multi-party setting, $n > 2$, $t < n/2$, and the computational parties are assumed to contribute no input and receive no output to ensure that they can be disjoint from the input and output parties. Then the input parties secret-share their inputs among the computational parties prior the protocol execution takes place and the output parties receive shares of the output and reconstruct the result after the protocol termination. This in particular means that, in order to comply with the above security definition, the computation used in protocol Π must be data-oblivious, which means that the sequence of operations and memory accesses used in Π must be independent of the input.

Performance of secure computation techniques is of grand significance, as protecting secrecy of data throughout the computation often incurs substantial computational costs. For that reason, besides security, efficient performance of the developed techniques is one of our primary goals. In both of our settings, computation of a linear combination of protected values can be performed locally by each participant (i.e., on encrypted values in the two-party setting and on secret-shared values in the multi-party setting), while multiplication is interactive. Because normally the

overhead of interactive operations dominates the runtime of a secure multi-party computation algorithm, its performance is measured in the number of interactive operations (such as multiplications, as well as other instances which include distributing shares of a private value or opening a secret-shared value in the multi-party setting). Furthermore, the round complexity, i.e., the number of sequential interactions, can have a substantial impact on the overall execution time, and serves as the second major performance metric. Lastly, in the two-party setting, public-key operations (and modulo exponentiations in particular) impose a significant computational overhead, and are used as an additional performance metric.

In this work, we use notation $[x]$ to denote that the value of x is protected, either through encryption or secret sharing.

5 BUILDING BLOCKS

Before presenting our solution, we give a brief description of the building blocks from the literature used in our solution. While the computation proceeds on floating point numbers, we first present building blocks that operate on integers followed by floating point operations.

5.1 Integer building blocks

First note that having secure implementations of addition and multiplication operations alone can be used to securely evaluate any functionality on protected values represented as an arithmetic circuit. Prior literature, however, concentrated on developing secure protocols for commonly used operations which are more efficient than general techniques. In particular, prior literature contains a large number of publications for secure computation on integers such as comparisons, bit decomposition, and other operations. From all of the available techniques, we have chosen the building blocks that yield the best performance for our construction, and are listed below.

Note that we use the following complexity for elementary arithmetic operations: addition and subtraction of two protected values in either two-party or multi-party setting involves no communication. Multiplication in the multi-party setting requires a computational party to send values to all other parties and wait for values from them, where all parties communicate their values simultaneously. This is treated as an elementary interactive operation in a single round. In the two-party setting, multiplication of two encrypted values $\mathsf{Enc}_{pk}(a)$ and $\mathsf{Enc}_{pk}(b)$ is computed interactively, during which one party chooses a random value r, forms $\mathsf{Enc}_{pk}(a-r)$ using homomorphic properties of the encryption scheme, helps the second party to decrypt $a-r$, after which the parties locally compute $\mathsf{Enc}_{pk}(br)$ and $\mathsf{Enc}_{pk}(b(a-r))$, respectively, and exchange the ciphertexts to obtain $\mathsf{Enc}_{pk}(ba)$. This involves two sequential messages, which we count as 2 rounds, and a small constant number of modulo exponentiations. This two-party implementation of the multiplication operation is used in computing the complexity of the building blocks that follow.

- $[c] \leftarrow \mathsf{XOR}([a],[b])$ computes exclusive OR of bits a and b as $[a]+[b]-2[a][b]$ using one multiplication.
- $[c] \leftarrow \mathsf{OR}([a],[b])$ computes OR of bits a and b as $[a]+[b]-[a][b]$.
- $[r] \leftarrow \mathsf{RandInt}(\ell)$ allows the parties to generate a random ℓ-bit value $[r]$ without any interaction. In the two-party setting, the participants produce an encryption of r, and in the multi-party setting, the parties use what can be viewed as a distributed pseudo-random function to produce shares of r (see, e.g., (Catrina and Saxena, 2010)).
- $[r] \leftarrow \mathsf{RandBit}()$ allows the parties to produce shares of a random bit $[r]$ using one interactive operation in the multi-party setting and using one multiplication (more precisely, by executing the XOR of two bits) in the two-party setting.
- $[b] \leftarrow \mathsf{Inv}([a])$ computes $b = a^{-1}$ (in the respective group or field). For a non-zero a this operation can be implemented by creating a random $[r]$, computing and opening $c = ra$, and setting $[b] = c^{-1}[r]$.
- $([y_1],\ldots,[y_\ell]) \leftarrow \mathsf{PreMul}([x_1],\ldots,[x_\ell])$ computes prefix-multiplication, where on input a sequence of integers $x_1,\ldots,x_\ell$, the output consists of values $y_1,\ldots,y_\ell$, where each $y_i = \prod_{j=1}^{i} x_j$. The most efficient implementation of this operation in our framework that we are aware of is from (Catrina and de Hoogh, 2010) that works only on non-zero elements, which is sufficient for our purposes.
- $([y_1],\ldots,[y_\ell]) \leftarrow \mathsf{PreOR}([x_1],\ldots,[x_\ell])$ computes prefix-OR of n input bits $x_1,\ldots,x_\ell$ and outputs $y_1,\ldots,y_\ell$ such that each $y_i = \bigvee_{j=1}^{i} x_j$, which we assume is implemented as in (Catrina and de Hoogh, 2010).
- $[b] \leftarrow \mathsf{EQ}([x],[y],\ell)$ is an equality protocol that on two ℓ-bit inputs x and y outputs a bit b which is set to 1 iff $x = y$. We use a secure implementation of this operation from (Catrina and de Hoogh, 2010), which is built using another protocol $[b] \leftarrow \mathsf{EQZ}([x'],\ell)$ that outputs bit $b = 1$ iff

$x' = 0$ by calling $\mathsf{EQZ}([x]-[y],\ell)$. EQ and EQZ thus have the same complexity for the same ℓ.

- $[b] \leftarrow \mathsf{LT}([x],[y],\ell)$ is a comparison protocol that on input two secret-shared ℓ-bit values x and y outputs a bit b which is set to 1 iff $x < y$. For the multi-party setting, we use the protocol from (Catrina and de Hoogh, 2010) and for the two-party setting, we use the protocol from (Kerschbaum et al., 2009).
- $[r] \leftarrow \mathsf{TruncPR}([x],\ell,k)$ computes $\lfloor [x]/2^k \rfloor$, where ℓ is the bitlength of x, with an additive error $c \leq 1$. When such an error is permissible, the protocol is significantly faster than precise truncation. In the two-party setting, (Dahl et al., 2012) proposes a constant-work protocol, while (Catrina and de Hoogh, 2010) provides the most efficient implementation in the multi-party setting.
- $[y] \leftarrow \mathsf{Trunc}([x],\ell,k)$ computes $\lfloor [x]/2^k \rfloor$ for an ℓ-bit x. This operation can be efficiently implemented using, for example, the techniques of (Catrina and de Hoogh, 2010) in the multi-party setting. In the two-party setting, the most efficient way of implementing the functionality is by executing $[y] \leftarrow \mathsf{TruncPR}([x],\ell,k)$, then correcting the error (if any) by comparing x and $y \cdot 2^k$ by calling $[c] \leftarrow \mathsf{LT}([x],2^k[y])$, and setting the result to $[y]-[c]$.
- $[x_{k-1}],\ldots,[x_0] \leftarrow \mathsf{BitDec}([x],\ell,k)$ performs bit decomposition of k least significant bits of x, where ℓ is the size of x. An efficient implementation of this functionality in both settings can be found in (Catrina and Saxena, 2010), the complexity of which is independent of the bitlength of x ℓ.
- $[2^x] \leftarrow \mathsf{Pow2}([x],\ell)$ raises 2 in the (unknown) power x supplied as the first argument, where the second argument ℓ specifies the bitlength of the representation. To ensure that 2^x can be represented using ℓ bits, the value of x is expected to be in the range $[0,\ell)$. We use an efficient implementation of this protocol from (Aliasgari et al., 2013).

Many of these protocols are used as building blocks in floating point protocols as opposed to being used directly in our solution. The complexities of these protocols in the two-party and multi-party settings are provided in Tables 1 and 2, respectively. In Table 1, notation C denotes the ciphertext length in bits, and notation D denotes the length of the auxiliary decryption information, which when sent by one of the parties allows the other party to decrypts a ciphertext. Communication is measured in bits, and computation is measured in modulo exponentiations. We list computational overhead incurred by each party separately, with the smaller amount of work first (which can be carried out by a client) followed by the larger amount of work (which can be carried out by a server).

5.2 Floating point building blocks

For floating point operation we adopt the same floating point representation that was used in (Aliasgari et al., 2013). Namely, a real number x is represented as 4-tuple $\langle v,p,s,z\rangle$, where v is an ℓ-bit normalized significand (i.e., the most significant bit of v is 1), p is a k-bit exponent, z is a bit that indicates whether the value is zero, and s is a bit set only when the value is negative. We obtain that $x = (1-2s)(1-z)v \cdot 2^p$. As in (Aliasgari et al., 2013), when $x = 0$, we maintain that $z = 1$, $v = 0$, and $p = 0$.

The work (Aliasgari et al., 2013) provides a number of secure floating point protocols, some of which we use in our solution as floating point building blocks. While the techniques of (Aliasgari et al., 2013) also provide the capability to detect and report errors (e.g., in case of division by 0, overflow or underflow, etc.), for simplicity of presentation, we omit error handling in this work. The building blocks from (Aliasgari et al., 2013) that we use here are:

- $\langle [v],[p],[z],[s]\rangle \leftarrow \mathsf{FLMul}(\langle [v_1],[p_1],[z_1],[s_1]\rangle, \langle [v_2],[p_2],[z_2],[s_2]\rangle)$ performs floating point multiplication of its two real valued arguments.
- $\langle [v],[p],[z],[s]\rangle \leftarrow \mathsf{FLDiv}(\langle [v_1],[p_1],[z_1],[s_1]\rangle, \langle [v_2],[p_2],[z_2],[s_2]\rangle)$ allows the parties to perform floating point division using $\langle [v_1],[p_1],[z_1],[s_1]\rangle$ as the dividend and $\langle [v_2],[p_2],[z_2],[s_2]\rangle$ as the divisor.
- $\langle [v],[p],[z],[s]\rangle \leftarrow \mathsf{FLAdd}(\langle [v_1],[p_1],[z_1],[s_1]\rangle, \langle [v_2],[p_2],[z_2],[s_2]\rangle)$ performs the computation of addition (or subtraction) of two floating point arguments.
- $[b] \leftarrow \mathsf{FLLT}(\langle [v_1],[p_1],[z_1],[s_1]\rangle, \langle [v_2],[p_2],[z_2],[s_2]\rangle)$ produces a bit, which is set to 1 iff the first floating point argument is less than the second argument.
- $\langle [v],[p],[z],[s]\rangle \leftarrow \mathsf{FLExp2}(\langle [v_1],[p_1],[z_1],[s_1]\rangle)$ computes the floating point representation of exponentiation $[2^x]$, where $[x] = (1-2[s_1])(1-[z_1])[v_1]2^{[p_1]}$.

These protocols were given in (Aliasgari et al., 2013) only for the multi-party setting, but we also evaluate their performance in the two-party setting using the previously listed integer building blocks. The complexities of these floating point protocols in both settings are reported in Tables 1 and 2.

Table 1: Complexity of building blocks in the two-party setting.

Protocol	Rounds	Communication size	Computation complexity	
			Client	Server
Mul	2	$3C+D$	2	3
Inv	1	$2C+2D$	4	4
TruncPR	2	$3C+D$	1	3
PreMul	5	$6\ell D+11\ell C$	$13\ell-4$	$16\ell-5$
PreOR	6	$6\ell D+13\ell C$	$16\ell-7$	$19\ell-8$
LT	2	$D+6C$	4	5
EQZ	6	$(6\log\ell+1)D+13\log\ell C$	$\ell+14\log\ell+3$	$\ell+17\log\ell+2$
Trunc	4	$2D+9C$	5	8
BitDec	$\log k+2$	$(k\log k+1)D+(3k\log k+2k+1)C$	$2k\log k+k+2$	$3k\log k+k+2$
Pow2	$\log\log\ell+7$	$(6\log\ell+(\log\log\ell)\log\ell+1)D+(3\log\ell\log\log\ell+13\log\ell+1)C$	$2\log\ell\log\log\ell+14\log\ell-2$	$3\log\ell\log\log\ell+17\log\ell-3$
FLMul	13	$10D+39C$	17	25
FLDiv	$2\log\ell+8$	$4\log\ell(D+3C)+4D+18C$	$6\log\ell+12$	$12\log\ell+16$
FLAdd	$\log\ell+45+\log\log\ell$	$(\ell\log\ell+14\log\ell+\log\ell(\log\log\ell)+6\log k+54)D+(15\ell+3\ell\log\ell+19\log\ell+13\log k+3(\log\ell)\log\log\ell+155)C$	$18\ell+k+2\ell\log\ell+32\log\ell+14\log k+2\log\ell\log\log\ell+125$	$21\ell+k+3\ell\log\ell+40\log\ell+17\log k+3\log\ell\log\log\ell+144$
FLLT	10	$(6\log\ell+20)D+(13\log\ell+63)C$	$k+14\log k+41$	$k+17\log k+57$
FLExp2	$15\log\ell+38$	$(10\ell+\ell\log\ell+12\log\ell+\log\ell\log\log\ell+35)D+(34\ell+3\ell\log\ell+26\log\ell+3(\log\ell)\log\log\ell+107)C$	$40\ell+2\ell\log\ell+28\log\ell+2\log\ell\log\log\ell+44$	$53\ell+3\ell\log\ell+3\log\ell\log\log\ell+34\log\ell+59$

5.3 Improved exponentiation

Floating point exponentiation is the most expensive operation used in the computation of our target HMM functionality. Furthermore, because each element of the HMM's output probability matrix is derived using GMM formula that involves exponentiation, the exponentiation protocol must be executed a large number of times. This means that any improvement in the performance of this operation results in a faster execution of the overall computation. For that reason, in this section, we describe a modified solution for the exponentiation operation that improves performance of the solution in (Aliasgari et al., 2013).

As mentioned in (Aliasgari et al., 2013), the major overhead of FLExp2 is caused by a sub-protocol FLProd, which computes the product of ℓ floating point values. FLProd accounts for 90% of FLExp2's overall time and we therefore focus on improving performance of this functionality. The original implementation of FLProd in (Aliasgari et al., 2013) executes $\ell-1$ floating point FLMul in a tree like fashion using a logarithmic number of rounds. In each round, FLProd calls FLMul on all pairs of available values in parallel and passes the result to the next round. FLMul is given below. The comparison and the second truncation are to ensure that the output's significand is a normalized ℓ-bit value, where ℓ is the bitlength of the significand's representation and the number of inputs to FLProd.

$\langle[v],[p],[z],[s]\rangle \leftarrow \mathsf{FLMul}(\langle[v_1],[p_1],[z_1],[s_1]\rangle, \langle[v_2],[p_2],[z_2],[s_2]\rangle)$

1. $[v] \leftarrow [v_1][v_2]$;
2. $[v] \leftarrow \mathsf{Trunc}([v], 2\ell, \ell-1)$;
3. $[b] \leftarrow \mathsf{LT}([v], 2^{\ell}, \ell+1)$;
4. $[v] \leftarrow \mathsf{Trunc}(2[b][v]+(1-[b])[v], \ell+1, 1)$;
5. $[z] \leftarrow \mathsf{OR}([z_1],[z_2])$;
6. $[s] \leftarrow \mathsf{XOR}([s_1],[s_2])$;
7. $[p] \leftarrow ([p_1]+[p_2]+\ell-[b])(1-[z])$;
8. return $\langle[v],[p],[z],[s]\rangle$;

Our main idea consists of changing FLProd by letting the intermediate values slightly grow in each round and removing the extra bits only after the final round of multiplication. This allows us to remove the comparison and the second truncation from FLMul. Note that the performance improvement of this modification is pronounced only in the multi-party setting because comparisons are very efficient in the two-party setting. We call this modified multiplication protocol as simplified floating point multiplication, SFLMul. SFLMul performs a simple multiplication of two significands and truncates the result by $\ell-1$ bits. This means the output significand is of length either $\gamma+1$ or γ bits when the input significands were of size γ. In the first round of FLProd, γ is equal to ℓ. In round i, for $1 \leq i \leq \lceil \log \ell \rceil$, the maximum bitlength is $\gamma = \ell + 2^i - 1$. To be able to multiply significands in this round, we want to be able to represent values larger than $2\ell+2^i-2$ bits and the solution to work correctly on this larger length representation. When ℓ is a power of 2, which we assume for the simplicity of exposition, $2\ell+2^{\lceil \log \ell \rceil}-2 = 3\ell-2$, which dictates our choice of parameters in SFLMul protocol below.

Table 2: Complexity of building blocks in the multi-party setting.

Protocol	Rounds	Interactive operations
Mul	1	1
TruncPR	2	$k+1$
PreMul	2	$3n-1$
PreOR	3	$5n-1$
LT	4	$4\ell-2$
EQZ	4	$\ell+4\log\ell$
Trunc	4	$4k+1$
BitDec	$\log k$	$k\log k$
Pow2	$\log\log\ell+1$	$\log\ell(\log\log\ell)+3\log\ell-1$
Trunc	$\log\log\ell+9$	$12\ell+(\log\ell)\log\log\ell+3\log\ell$
FLMul	11	$8\ell+10$
FLDiv	$2\log\ell+7$	$2\log\ell(\ell+2)+3\ell+8$
FLAdd	$\log\ell+\log\log\ell+27$	$14\ell+9k+(\log\ell)\log\log\ell+(\ell+9)\log\ell+4\log k+37$
FLLT	6	$4\ell+5k+4\log k+13$
FLExp2	$12\log\ell+\log\log\ell+24$	$8\ell^2+11\ell+\ell\log\ell+(\log\ell)\log\log\ell+7\log\ell+16k+5$
New FLExp2	$6\log\ell+\log\log\ell+31$	$4\ell^2+23\ell+3\ell\log\ell+(\log\ell)\log\log\ell+6\log\ell+16k+1$

$[v] \leftarrow \mathsf{SFLMul}([v_1],[v_2])$

1. $[v] \leftarrow [v_1][v_2]$;
2. $[v] \leftarrow \mathsf{Trunc}([v], 3\ell-2, \ell-1)$;
3. return $[v]$;

After the last, $(\log\ell)$th, round of multiplication in FLProd, the resulting significand will have at most $2\ell-1$ bits, and the extra bits should be removed. This can be done by a normalization method similar to the one used in (Aliasgari et al., 2013), where we first bit-decompose the value, compute the most significant non-zero bit, and truncate the result to retain ℓ bits starting from the most significant non-zero bit. We also need to adjust the resulting exponent by the number of truncated bits. Our new FLProd protocol is given next. Because it is used for exponentiation with positive non-zero values, we disregard (constant) zero and sign bits, and operate on significands $[v_i]$ and exponents $[p_i]$.

$\langle [v],[p]\rangle \leftarrow \mathsf{FLProd}(\langle [v_1],[p_1]\rangle,\ldots,\langle [v_\ell],[p_\ell]\rangle)$

1. for $i=1$ to $\log\ell$ do
2. for $j=1$ to $\ell/2^i$ do in parallel
3. $[v_j] \leftarrow \mathsf{SFLMul}([v_{2j-1}],[v_{2j}])$;
4. $[u_{2\ell-2}],\ldots,[u_0] \leftarrow \mathsf{BitDec}([v_1], 2\ell-1, 2\ell-1)$;
5. $[h_0],\ldots,[h_{2\ell-2}] \leftarrow \mathsf{PreOR}([u_{2\ell-2}],\ldots,[u_0])$;
6. $[p_0] \leftarrow 2\ell-1-\sum_{i=0}^{2\ell-2}[h_i]$;
7. $[2^{p_0}] \leftarrow 1+\sum_{i=0}^{2\ell-2} 2^i(1-[h_i])$;
8. $[v] \leftarrow \mathsf{Trunc}([2^{p_0}][v], 2\ell-1, \ell-1)$;
9. $[p] \leftarrow \Sigma_{i=1}^{\ell}[p_i]-[p_0]+\ell(\ell-1)$;
10. return $\langle [v],[p]\rangle$;

The increase in the size of the intermediate values from the original $2\ell+1$ to $3\ell-2$ requires that the representation of the values is large enough to preserve correctness of the computation. In the two-party setting, this does not require any additional provisions as the group size over which the computation takes place is significantly larger than $2^{3\ell}$ for any practical value of ℓ. In the multi-party setting, however, the field size should be chosen appropriately to accommodate the need to represent larger values. This means that we need to increase the bitlength $|p|$ from $> 2\ell+1+\sigma$ to $> 3\ell-2+\sigma$, where σ is the security parameter for achieving information-theoretical statistical security. A larger field size yields slower execution and thus decreased performance. To avoid this issue, we utilize the idea of field and the corresponding share conversion proposed in (Damgård and Thorbek, 2008; Cramer et al., 2005). All computation prior and after FLProd is performed with the field size originally suggested in (Aliasgari et al., 2013), while the FLProd protocol is executed using a larger field. Fortunately, the share-conversion from one field to another can in our case be performed non-interactively with no communication cost.

6 SECURE VITERBI AND GMM COMPUTATION

Now we are ready to put everything together and describe our privacy-preserving solution for HMM and GMM computation based on Viterbi algorithm using floating point numbers.

To execute the HMM algorithm given in section 3, we first need to perform GMM computation to derive the output probabilities b_{jk} using equation 1. It was suggested in (Smaragdis and Shashanka, 2007) that the ith components of a GMM, $g_i(x) = -\frac{1}{2}(x-$

$\mu_i)^T\Sigma_i^{-1}(x-\mu_i)$, is represented as $x^TY_ix+y_i^Tx+y_{i0}$, where $Y_i=-\frac{1}{2}\Sigma_i^{-1}$, $y_i=\Sigma_i^{-1}\mu_i$, and $y_{i0}=-\frac{1}{2}\mu_i^T\Sigma_i^{-1}\mu_i$. The suggested representation increases the number of additions and multiplications than the original formula and therefore would result in a slower performance. In particular, because FLAdd is a relatively expensive protocol, we would like to minimize the use of this function. Thus we suggest that the parties first subtract μ_i from the observed vector x and engage in matrix multiplication to compute g_i. Note that the parties can run the above computation in parallel for all values of i, j, and k in equation 1. After its completion, the parties proceed with performing the secure version of the Viterbi algorithm.

The Viterbi algorithm requires (floating point) multiplication, max, and arg max. The multiplication is implemented using FLMul, while the max and arg max operations can be implemented using FLLT in a tree like fashion (i.e., we first compare every two adjacent elements, then every two maximum elements from the first round of comparisons, etc.). To perform arg max of two floating point numbers at indices i and j, let $[b]$ be the outcome of the FLLT operation applied to the numbers at these indices. Then we set arg max to be equal to $[b]j+(1-[b])i$. Therefore, arg max of a number of floating point values can be computed in a tree like fashion using the above method for each comparison.

After completing the recursion step of the Viterbi algorithm, we need to retrieve the sequence of states in the HMM that resulted in the most likely path (lines b and c of the termination step in the Viterbi algorithm). If the sequence of these states can be made publicly available, then the parties open the values corresponding to q_i^* for $1\leq i\leq T$ to learn the sequence. However, in a more likely event that this sequence needs to stay protected from one or more parties, we make use of the protocol Pow2 from (Aliasgari et al., 2013). To compute $[q_t^*]=\psi_{t+1}([q_{t+1}^*])$ given $[q_{t+1}^*]$, we execute $\mathsf{BitDec}(\mathsf{Pow2}([q_{t+1}^*]-1,N),N,N)$ that will produce N bits, all of which are 0 except the bit at position q_{t+1}^*. We then multiply each bit of the result by the respective element of the vector ψ_{t+1} and add the resulting values to obtain $[q_t^*]$.

In the two-party setting, however, one of the parties (e.g., the server) will learn the sequence as its output and a more efficient approach is possible. The idea is to take advantage of the fact that all encrypted values of the matrix ψ are held by both parties. In this case, the party receiving the output retrieves $\mathsf{Enc}_{pk}(\psi_{t+1}(q_{t+1}^*))$ using its knowledge of q_{t+1}^* which became available to that party in the previous step, randomizes the ciphertext by multiplying it with a fresh encryption of 0, and sends the result to the other party. Note that this randomization does not change the value of the underlying plaintext, but makes it such that the party receiving it cannot link the randomized ciphertext to one of the encryptions it possesses. This party then applies decryption to the received ciphertext and sends it back to the receiving party, who finishes the decryption, learns q_t^*, and continues to the next iteration of the computation.

The security of the proposed solution can be stated as follows:

Theorem 1. *The Viterbi solution above is secure both in the* two-party *and* multi-party *settings in the semi-honest security model.*

Proof sketch. The security of our solution in the semi-honest model with respect to Definition 1 is based on the fact that we only combine previously known secure building blocks. Such building blocks take protected inputs and produce protected outputs, which means that their composition does not reveal information about private values. In particular, we can apply Cannetti's composition theorem (Canetti, 2000), which states that a composition of secure subprotocols leads to security of the overall solution, to arrive at security of the overall solution. More formally, in both two-party and multi-party settings, we can build a simulator S of the overall solution according to Definition 1, which without access to private data produces a view that cannot be distinguished from the participants' views in the real protocol execution. Our simulator calls the corresponding simulators for the underlying building blocks. Then because each underlying simulator produces a view that is either computationally or statistically indistinguishable (depending on the setting) from the view of a particular party and no information is revealed while combining the building blocks, the simulation of each participant's view in the overall protocol also cannot be distinguished from a real protocol execution. We thus obtain security of the solution in the semi-honest model. □

The security of our solution in the multi-party setting can also be extended to the malicious security model. In that case, to show security in presence of malicious adversaries, we need to ensure that (i) all participants prove that each step of their computation was performed correctly and that (ii) if some dishonest participants quit, others will be able to reconstruct their shares and proceed with the rest of the computation. The above is normally achieved using a verifiable secret sharing scheme (VSS), and a large number of results have been developed over the years (e.g., (Gennaro et al., 1998; Hirt and Maurer, 2001;

Beerliova-Trubiniova and Hirt, 2008; Damgård et al., 2008; Damgård et al., 2010) and others). In particular, because any linear combination of shares is computed locally, each participant is required to prove that it performed each multiplication correctly on its shares. Such results normally work for $t < \frac{n}{3}$ in the information theoretic or computational setting with different communication overhead and under a variety of assumptions about the communication channels. Additional proofs associated with this setting include proofs that shares of a private value were distributed correctly among the participants (when the dealer is dishonest) and proofs of proper reconstruction of a value from its shares (when not already implied by other techniques). In addition, if at any point of the computation the participants are required to input values of a specific form, they would have to prove that the values they supplied are well formed. Such proofs are needed by the implementations of some of the building blocks (e.g., RandInt).

Thus, security of our protocols in the malicious model in the multi-party setting can be achieved by using standard VSS techniques, e.g., (Gennaro et al., 1998; Cramer et al., 2000), where a range proof, e.g., (Peng and Bao, 2010) will be additionally needed for the building blocks. These VSS techniques would also work with malicious input parties (who distribute inputs among the computational parties), who would need to prove that they generate legitimate shares of their data.

To show security of our solution in presence of malicious adversaries in the two-party setting, we likewise need to show that the participants perform all operations correctly. Because Paillier encryption is an additively homomorphic encryption that can be used as a threshold encryption scheme, we can employ existing zero-knowledge proofs of knowledge for Paillier ciphertexts. Example existing proofs for Paillier encryption include a proof of knowledge of plaintext (Damgård and Jurik, 2001; Baudron et al., 2001), a proof that two plaintexts are equal (Baudron et al., 2001), a proof that a ciphertext encrypts one value from a given set (Damgård and Jurik, 2001; Baudron et al., 2001), a proof that a ciphertext encrypts a product of two other given encrypted values (Cramer et al., 2001), and a range proof for the exponent a of plaintext b^a (Lipmaa et al., 2002). We leave the investigation of how these and possibly newly designed zero-knowledge proofs can be used to achieve security of our solution in the malicious model in the two-party setting to the full version of this work.

Lastly, we defer experimental results that empirically evaluate performance of the developed solution to the full version of this work.

7 CONCLUSIONS

In this work, we treat the problem of privacy-preserving Hidden Markov models computation which is commonly used for many applications including speaker recognition. We develop the first provably secure techniques for HMM's Viterbi and GMM computation using floating point arithmetic. Our solutions are designed for both two-party computation that utilizes homomorphic encryption and multi-party computation based on secret sharing, which cover a wide variety of real-life settings, and are designed to minimize their overhead.

ACKNOWLEDGMENTS

This work was supported in part by grants CNS-1223699 from the National Science Foundation and FA9550-13-1-0066 from the Air Force Office of Scientific Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Aliasgari, M., Blanton, M., Zhang, Y., and Steele, A. (2013). Secure computation on floating point numbers. In *Network and Distributed System Security Symposuim (NDSS)*.

Baudron, O., Fouque, P.-A., Pointcheval, D., Stern, J., and Poupard, G. (2001). Practical multi-candidate election scheme. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 274–283.

Beerliova-Trubiniova, Z. and Hirt, M. (2008). Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference (TCC)*, pages 213–230.

Canetti, R. (2000). Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202.

Catrina, O. and de Hoogh, S. (2010). Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks (SCN)*, pages 182–199.

Catrina, O. and Saxena, A. (2010). Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security (FC)*, pages 35–50.

Cramer, R., Damgård, I., and Ishai, Y. (2005). Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography Conference (TCC)*, pages 342–362.

Cramer, R., Damgård, I., and Maurer, U. (2000). General secure multi-party computation from any linear

secret-sharing scheme. In *Advances in Cryptology – EUROCRYPT*, pages 316–334.

Cramer, R., Damgård, I., and Nielsen, J. (2001). Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology – EUROCRYPT*, pages 280–289.

Dahl, M., Ning, C., and Toft, T. (2012). On secure two-party integer division. In *Financial Cryptography and Data Security (FC)*, pages 164–178.

Damgård, I., Ishai, Y., and Krøigaard, M. (2010). Perfectly secure multiparty computation and the computational overhead of cryptography. In *Advances in Cryptology – EUROCRYPT*, pages 445–465.

Damgård, I., Ishai, Y., Krøigaard, M., Nielsen, J., and Smith, A. (2008). Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology – CRYPTO*, pages 241–261.

Damgård, I. and Jurik, M. (2001). A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *International Workshop on Practice and Theory in Public Key Cryptography (PKC)*, pages 119–136.

Damgård, I. and Thorbek, R. (2008). Efficient conversion of secret-shared values between different fields. ePrint Archive Report 2008/221.

Franz, M., Deiseroth, B., Hamacher, K., Jha, S., Katzenbeisser, S., and Schröder, H. (2012). Towards secure bioinformatics services (short paper). In *Financial Cryptography and Data Security (FC)*, pages 276–283. Springer.

Gennaro, R., Rabin, M., and Rabin, T. (1998). Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 101–111.

Hirt, M. and Maurer, U. (2001). Robustness for free in unconditional multi-party computation. In *Advances in Cryptology – CRYPTO*, pages 101–118.

Kerschbaum, F., Biswas, D., and de Hoogh, S. (2009). Performance comparison of secure comparison protocols. In *International Workshop on Database and Expert Systems Application (DEXA)*, pages 133–136.

Lipmaa, H., Asokan, N., and Niemi, V. (2002). Secure Vickrey auctions without threshold trust. In *Financial Cryptography (FC)*, pages 87–101.

Nguyen, H. and Roughan, M. (2012a). Multi-observer privacy-preserving hidden markov models. In *Network Operations and Management Symposium (NOMS)*, pages 514–517.

Nguyen, H. and Roughan, M. (2012b). On the identifiability of multi-observer hidden markov models. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1873–1876.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – EUROCRYPT*, pages 223–238.

Pathak, M., Portelo, J., Raj, B., and Trancoso, I. (2012). Privacy-preserving speaker authentication. *Information Security Conference (ISC)*, pages 1–22.

Pathak, M. and Raj, B. (2011). Privacy preserving speaker verification using adapted gmms. In *Interspeech*, pages 2405–2408.

Pathak, M., Rane, S., Sun, W., and Raj, B. (2011). Privacy preserving probabilistic inference with hidden Markov models. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5868–5871.

Peng, K. and Bao, F. (2010). An efficient range proof scheme. In *IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT)*, pages 826–833.

Rabiner, L. (1989). A tutorial on hidden Markov-models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Shamir, A. (1979). How to share a secret. *Communications of the ACM*, 22(11):612–613.

Shashanka, M. (2010). A privacy preserving framework for gaussian mixture models. In *IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 499–506. IEEE.

Smaragdis, P. and Shashanka, M. (2007). A framework for secure speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 15(4):1404–1413.

APPENDIX

SECURITY WEAKNESS OF (Pathak and Raj, 2011)

Here we show that the logsum protocol used in (Pathak and Raj, 2011) for HMM computation has a security flaw, which demonstrates the importance of rigorous analysis of protocols with respect to the level of security they can offer.

The input to the two-party logsum protocol in (Pathak and Raj, 2011) consists of N encrypted logarithms of private values x_i. During the protocol, one party, Alice, learns products $e^r x_i$ for all inputs and single random r. Alice thus knows the ratios of the private values. These ratios reveal a substantial amount of information about the secret values, and in particular the relative magnitude of the inputs, no information about which should be revealed. This information leakage can lead to more significant or even full data recovery if these private values are used in other operations and a function of them is known or if an outside information about at least one of the x_i's is available (e.g., if one of them comes from Alice). For example, if Alice knows only one x_i, she will be able to recover all remaining private values. This clearly undermines the security of the designed system and is unacceptable for a security solution.